STREAM: The Stanford Stream Data Manager

The STREAM Group[*]

Stanford University

`http://www-db.stanford.edu/stream`

Abstract

The STREAM *project at Stanford is developing a general-purpose system for processing continuous queries over multiple continuous data streams and stored relations. It is designed to handle high-volume and bursty data streams with large numbers of complex continuous queries. We describe the status of the system as of early 2003 and outline our ongoing research directions.*

1 Introduction

The *STanford stREam datA Manager (STREAM)* project at Stanford is developing a general-purpose *Data Stream Management System (DSMS)* for processing continuous queries over multiple continuous data streams and stored relations. The following two fundamental differences between a DSMS and a traditional DBMS have motivated us to design and build a DSMS from scratch:

1. A DSMS must handle multiple continuous, high-volume, and possibly time-varying *data streams* in additional to managing traditional stored relations.

2. Due to the continuous nature of data streams, a DSMS needs to support long-running *continuous queries*, producing answers in a continuous and timely fashion.

A high-level view of STREAM is shown in Figure 1. On the left are the incoming *Input Streams*, which produce data indefinitely and drive query processing. Processing of continuous queries typically requires intermediate state, which we denote as *Scratch Store* in the figure. This state could be stored and accessed in memory or on disk. Although we are concerned primarily with the online processing of continuous queries, in many applications stream data also may be copied to an *Archive*, for preservation and possible offline processing of expensive analysis or mining queries. Across the top of the figure we see that users or applications register *Continuous Queries*, which remain active in the system until they are explicitly deregistered. Results of continuous queries are generally transmitted as output data streams, but they could also be relational results that are updated over time (similar to materialized views).

[*]Active members as of March 2003: A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, J. Widom. This work was supported by NSF Grants IIS-0118173 and IIS-9817799, by Stanford Graduate Fellowships from 3Com, Rambus, and Sequoia Capital, by a Siebel Scholarship, and by support from the Okawa Foundation, Microsoft, Stanford Networking Research Center, and Veritas. Nishizawa is visiting Stanford from Hitachi, Ltd.

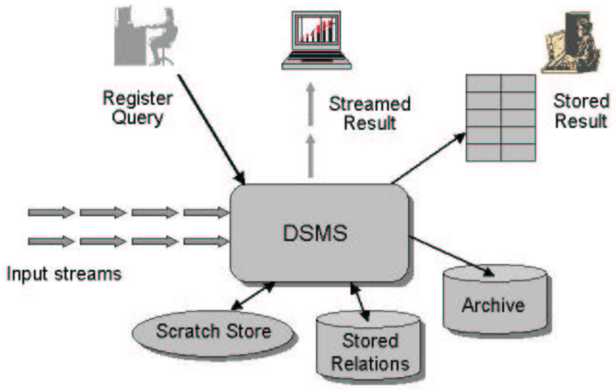

Figure 1: Overview of STREAM

Currently STREAM offers a Web system interface through direct HTTP, and we are planning to expose the system as a Web service through SOAP. Thus, remote applications can be written in any language and on any platform. Applications can register queries and receive the results of a query as a streaming HTTP response in XML. To allow interactive use of the system, we have developed a Web-based GUI as an alternative way to register queries and view results, and we provide an interactive interface for visualizing and modifying system behavior (see Section 4).

In Sections 2 (Query Language and Processing), 3 (Operator Scheduling), and 4 (User Interface) we describe the most important components of STREAM. In Section 5 we outline our current research directions. Due to space limitations this paper does not include a section dedicated to related work. We refer the reader to our recent survey paper [BBD$^+$02], which provides extensive coverage of related work.

2 Query Language and Processing

We first describe the query language and semantics for continuous queries supported by STREAM. The latter half of this section describes STREAM's query processing architecture.

2.1 Query Language and Semantics

We have designed an abstract semantics and a concrete declarative query language for continuous queries over data streams and relations. We model a *stream* as an unbounded, append-only bag of ⟨*tuple*, *timestamp*⟩ pairs, and a *relation* as a time-varying bag of tuples supporting updates and deletions as well as insertions. Our semantics for continuous queries over streams and relations leverages well-understood relational semantics. Streams are converted into relations using special *windowing* operators; transformations on relations are performed using standard relational operators; then the transformed relational data is (optionally) converted back into a streamed answer. This semantics relies on three abstract building blocks:

1. A relational query language, which we can view abstractly as a set of relation-to-relation operators.
2. A *window specification language* used to extract tuples from streams, which we can view as a set of stream-to-relation operators. In theory these operators need not have anything to do with "windows," but

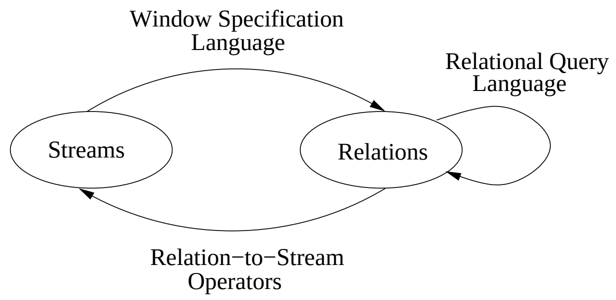
Window Specification Language
Relational Query Language
Streams
Relations
Relation−to−Stream Operators

Figure 2: Mappings used in abstract semantics

in practice windowing is the most common way of producing bounded sets of tuples from unbounded streams [BBD$^+$02].

3. A set of relation-to-stream operators.

The interaction among these three building blocks is depicted in Figure 2.

We have developed a concrete declarative query language, *CQL* (for *Continuous Query Language*), which instantiates our abstract semantics. Our language uses SQL as its relational query language, its window specification language is derived from SQL-99, and it includes three relation-to-stream operators. The CQL language also supports syntactic shortcuts and defaults for convenient and intuitive query formulation. The complete specification of our query semantics and CQL is provided in an earlier paper [ABW02]. The interested reader is referred to our *Stream Query Repository* [SQR], which contains queries from many realistic stream applications, including a large number and variety of queries expressed in CQL. A significant fraction of CQL has been implemented to date, as described in the next section.

2.2 Query Processing

When a continuous query specified in CQL is registered with STREAM, it is compiled into a *query plan*. The query plan is merged with existing query plans whenever possible, in order to share computation and memory. Alternatively, the structure of query plans can be specified explicitly using XML. A query plan in our system runs continuously and is composed of three different types of components:

1. Query *operators* correspond to the three types of operators in our abstract semantics (Section 2.1). Each operator reads tuples from a set of input queues, processes the tuples based on its semantics, and writes its output tuples into an output queue.

2. Inter-operator *queues* are used to buffer the output of one operator that is passed as input to one or more other operators. Incoming stream tuples and relation updates are placed in *input queues* feeding leaf operators.

3. *Synopses* maintain run-time state associated with operators.

STREAM supports the standard relational operators (including aggregation and duplicate elimination), *window* operators that compute time-based, tuple-based, and partitioned windows over streams [ABW02], three operators that convert relations into streams, and *sampling* operators for approximate query answering. Note that the queues and synopses for the active query plans in the system comprise the *Scratch Store* depicted in Figure 1.

A synopsis stores intermediate state at some operator in a running query plan, as needed for future evaluation of that operator. For example, a *sliding-window join operator* [KNV03] must have access to all the tuples that are

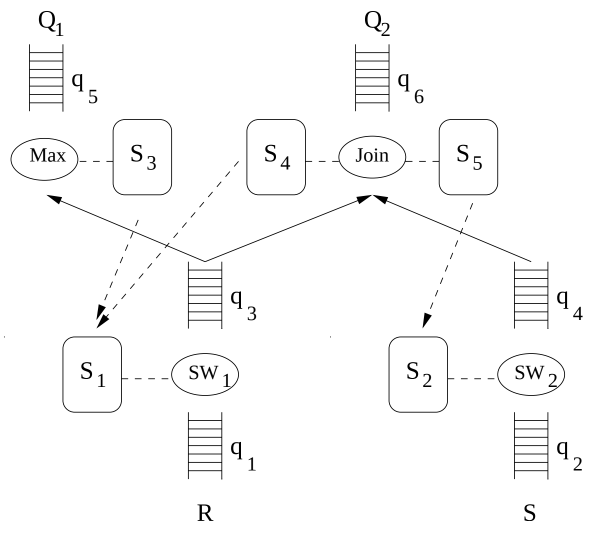

Figure 3: STREAM query plans

part of the current window on each of its input streams, so we maintain one *sliding-window synopsis* (typically a hash table) for each of these streams. On the other hand, simple filter operators, such as selection and duplicate-preserving projection, do not require a synopsis since they do not need to maintain state. The most common use of a synopsis in our system is to materialize a relation or a view (e.g., a sliding window). Synopses can also be used to store a summary of the tuples in a stream or a relation for approximate query answering. For this reason we have implemented *reservoir samples* [Vit85] over streams, and we will soon add *Bloom filters* [MW$^+$03].

Figure 3 illustrates plans for two queries, Q_1 and Q_2, over input streams R and S. Query Q_1 is a windowed-aggregate query: it maintains the maximum value of attribute $R.A$ over a sliding window on stream R. Query Q_2 is a sliding-window join query over streams R and S. Together the plans contain four operators $\texttt{SW}_1$, $\texttt{SW}_2$, `Max`, and `Join`, five synopses S_1–S_5, and six queues q_1–q_6. $\texttt{SW}_1$ is a sliding-window operator that reads stream R's tuples from queue q_1, updates the sliding-window synopsis S_1, and outputs the inserts and deletes to this sliding window into queue q_3. Thus, queue q_1 represents stream R, while queue q_3 represents the relation that is the sliding-window on stream R. Similarly, $\texttt{SW}_2$ processes stream S's tuples from q_2, updating synopsis S_2 and queue q_4. The `Max` operator maintains the maximum value of $R.A$ incrementally over the window on R, using the inserts and deletes from the window maintained by $\texttt{SW}_1$. Whenever the current maximum value expires from the window, `Max` will potentially need to access the entire window to compute the new maximum value. Thus, `Max` must materialize this window in its synopsis S_3. However, since S_3 is simply a time-shifted version of S_1, we can share the data store between S_1 and S_3, as indicated by the dotted arrow from S_3 to S_1. Similarly, the sliding-window synopsis S_4 maintained by the join operator `Join` can be shared with S_1 and S_3, and S_5 can be shared with S_2. Also, queue q_3 is shared by `Max` and `Join`, effectively sharing the window-computation subplan between queries Q_1 and Q_2.

The *Aurora* system [CCC$^+$02] supports shared queues, used to share storage for sliding windows on streams. Our system goes a step further in synopsis-sharing, including the ability to share the storage and maintenance overhead for indexes over the synopses as well. For example, if `Max` in Figure 3 computed `Group By` $R.B$, `Max` $R.A$, and `Join` used the join predicate $R.B = S.B$, then it would be useful to maintain a hash-index over $R.B$ in synopsis S_1 which both `Max` and `Join` could use. We currently support shared windows over streams where all the window specifications need not be the same, and shared materialized views, which are effectively common subexpressions in our query plans [CDTW00]. We use novel techniques to eliminate from synopses tuples that will not be accessed in the future, for example using reasoning based on constraints on the input streams [BW02].

Execution of query operators is controlled by a global *scheduler*, discussed next in Section 3. The operators have been implemented in such a way that they make no assumptions about the scheduling policy, giving the scheduler complete flexibility to adapt its scheduling strategy to the query workload and input stream characteristics.

3 Operator Scheduling

The execution of query plans is controlled by a global scheduler running in the same thread as all the operators in the system. (The I/O operations are handled by a separate thread.) Each time the scheduler is invoked, it selects an operator to execute and calls a specific procedure defined for that operator, passing as a parameter the maximum amount of time that the operator should run before returning control to the scheduler. The operator may return earlier if its input queues become empty.

The goals of a scheduler in a continuous query system are somewhat different than in a traditional DBMS. Some traditional scheduling objectives, such as minimizing run-time resource consumption and maximizing throughput, are applicable in the context of continuous queries, whereas other objectives, such as minimizing query response time, are not directly relevant in a continuous query setting, though they may have relevant counterparts (e.g., minimizing average latency of results). One objective that takes on unusual importance when processing data streams is careful management of run-time resources such as memory. Memory management is a particular challenge when processing streams because many real data streams are irregular in their rate of arrival, exhibiting burstiness and variation of data arrival rate over time. This phenomenon has been observed in networking [FP95], web-page access patterns, e-mail messages [Kle02], and elsewhere. When processing high-volume and bursty data streams, temporary bursts of data arrival are usually buffered, and this backlog of tuples is processed during periods of light load. However, it is important for the stream system to minimize the memory required for backlog buffering. Otherwise, total memory usage can exceed the available physical memory during periods of heavy load, causing the system to page to disk and limiting system throughput. To address this problem, we have developed an operator scheduling strategy that minimizes the memory requirement for backlog buffering [BBDM03]. This strategy, called *Chain scheduling*, is near-optimal in minimizing run-time memory usage for single-stream queries involving selections, projections, and foreign-key joins with stored relations. Chain scheduling also performs well for queries with sliding-window joins over multiple streams, and multiple queries of the above types.

The basic idea in Chain scheduling is to break up query plans into disjoint chains of consecutive operators based on their effectiveness in reducing run-time memory usage, favoring operators that “consume” a large number of tuples per time unit and “produce” few output tuples. This metric also determines the scheduling priority of each operator chain. Chain scheduling decisions are made by picking the operator chain with highest priority among those that have operators that are ready to execute and scheduling the first ready operator in that chain. Complete details of Chain, proofs of its near-optimality, and experimental results demonstrating the benefits of Chain with respect to other scheduling strategies, are provided in an earlier paper [BBDM03].

While Chain achieves run-time memory minimization, it may suffer from starvation and poor response times during bursts. As ongoing work, we are considering how to adapt our strategy to take into account these additional objectives.

4 User Interface

We are developing a comprehensive interactive interface for STREAM users, system administrators, and system developers to visualize and modify query plans as well as query-specific and system-wide resource allocation while the system is in operation.

4.1 Query Plan Execution

STREAM will provide a graphical interface to visualize the execution of any registered continuous query. Query plans are implemented as networks of *entities*, each of which is an operator, a queue, or a synopsis. The query plan execution visualizer will provide the following features.

1. View the structure of the plan and its component entities.
2. View specific attributes of an entity, e.g., the amount of memory being used by a synopsis in the plan.
3. View data moving through the plan, e.g., tuples entering and leaving inter-operator queues, and synopsis contents growing and shrinking as operators execute. Depending on the scope of activity individual tuples or tuple counts can be visualized.

4.2 Global System Behavior

'The query execution visualizer described in the previous section is useful for visualizing the execution and resource utilization of a single query, or a small number of queries that may share plan components. However, a system administrator or developer might want to obtain a more global picture of DSMS behavior. The STREAM system will provide an interface to visualize system-wide query execution and resource utilization information. The supported features include:

1. View the entire set of query plans in the system, with the level of detail dependent on the number and size of plans.
2. View the fraction of memory used by each query in the system, or in more detail by each queue and each synopsis.
3. View the fraction of processor time consumed by each query in the system.

4.3 Controlling System Behavior

Visualizing query-specific and system-wide execution and resource allocation information is important for system administrators and developers to understand and tune the performance of a DSMS running long-lived continuous queries. A sophisticated DSMS should adapt automatically to changing stream characteristics and changing query load, but it is still useful for "power users" and certainly useful for system developers to have the capability to control certain aspects of system behavior. STREAM does or will support the following features:

1. Run-time modification of memory allocation, e.g., increasing the memory allocated to one synopsis while decreasing memory for another.
2. Run-time modification of plan structure, e.g., changing the order of synopsis joins in a query over multiple streams, or changing the type of synopsis used by a join operator.
3. Run-time modification of the scheduling policy, choosing among several alternative policies.

5 Directions of Ongoing Research

This section outlines the problems that we are addressing currently in the STREAM project, in addition to implementing the basic prototype as described above. These problems fall broadly into the areas of efficient query processing algorithms, cost-based optimization and resource allocation, operator scheduling, graceful degradation under overload, and distributed stream processing.

Efficient query processing: Our system needs efficient query processing algorithms to handle high-volume data streams and large numbers of complex continuous queries. Some of the issues we are addressing in this area include techniques for sharing computation and memory aggressively among query plans, algorithms for multi-way sliding-window joins over streams, tradeoffs between incremental computation and recomputation for different types of continuous queries, and strategies for processing continuous queries efficiently while ensuring correctness in the absence of time-synchronization among stream sources and the DSMS.

Cost-based optimization and resource allocation: Although we have implemented support for a significant fraction of CQL in STREAM to date, our query plan generator is fairly naive and uses hard-coded heuristics to generate query plans. We are now moving towards one-time and dynamic cost-based query optimization of CQL queries. Since CQL uses SQL as its relational query language, we can leverage most of the one-time optimization techniques used in traditional relational DBMSs. Our unique optimization techniques include relocating window operators in query plans, exploiting stream constraints to reduce window sizes without affecting result correctness, and identifying opportunities for sharing computation (e.g., common subexpression computation, index maintenance) and memory (synopses and queues). Apart from choosing plans shapes and operators, a query optimizer must allocate resources such as memory within and across queries. One of the problems we are addressing in this area is how to allocate resources to query plans so as to maximize result precision whenever resource limitations force approximate query results. We are also exploring dynamic and adaptive approaches to query processing and resource allocation. Our adaptive query processing is less fine-grained than *Eddies* (as used in the *Telegraph* project [CC$^+$03]). Our approach relies on two interacting components: a *monitor* that captures properties of streams and system behavior, and an *optimizer* that can reconfigure query plans and resource allocation as properties change over time.

Scheduling: As described in Section 3, the Chain scheduling strategy achieves run-time memory minimization, but it may suffer from poor response times during bursts. As ongoing work, we are adapting Chain to minimize total run-time memory usage for queries under the constraint that the latency of any query-result tuple must not exceed a given threshold. Another planned extension needed for a complete scheduling strategy for a DSMS is the intelligent handling of query workloads where synopses and queues do not all fit into the physical memory available in the DSMS.

Graceful degradation under overload: There could be large intervals of time when input stream arrival rates exceed the maximum rate at which the DSMS can process its query workload over these streams. As shown by the *Aurora* system [CCC$^+$02], a general approach to handle such overload situations is *load shedding*. The system load is reduced to manageable levels by dropping input tuples selectively so that the overall *quality-of-service* given by the system degrades as little as possible [CCC$^+$02]. Ongoing work in our project adopts a similar approach, using sampling-based techniques to drop input tuples with the goal of minimizing the overall weighted error in query results incurred during overload situations.

Distributed stream processing: A final important aspect of our long-term research agenda is to incorporate distributed data stream processing techniques into the STREAM system. Data stream sources are frequently geographically dispersed, and our experiments and simulations show that processing strategies that take this fact into account can result in significant savings in computation and communication costs [OJW03, BO03]. We plan to modify STREAM to function in a distributed environment, incorporating specialized distributed data processing strategies.

References

[ABW02] A. Arasu, S. Babu, and J. Widom. An abstract semantics and concrete language for continuous queries over streams and relations. Technical report, Stanford University Database Group, November 2002. Available at http://dbpubs.stanford.edu/pub/2002-57.

[BBD$^+$02] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of the 2002 ACM Symp. on Principles of Database Systems*, pages 1–16, June 2002.

[BBDM03] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator scheduling for memory minimization in data stream systems. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, June 2003. (To appear).

[BO03] B. Babcock and C. Olston. Distributed top-k monitoring. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, June 2003. (To appear).

[BW02] S. Babu and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. Technical report, Stanford University Database Group, November 2002. Available at http://dbpubs.stanford.edu/pub/2002-52.

[CC$^+$03] S. Chandrasekaran, O. Cooper, et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. First Biennial Conf. on Innovative Data Systems Research (CIDR)*, January 2003.

[CCC$^+$02] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams–a new class of data management applications. In *Proc. 28th Intl. Conf. on Very Large Data Bases*, August 2002.

[CDTW00] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390, May 2000.

[FP95] S. Floyd and V. Paxson. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[Kle02] J. Kleinberg. Bursty and hierarchical structure in streams. In *Proc. of the 2002 ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, August 2002.

[KNV03] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Proc. of the 2003 Intl. Conf. on Data Engineering*, March 2003.

[MW$^+$03] R. Motwani, J. Widom, et al. Query processing, approximation, and resource management in a data stream management system. In *Proc. First Biennial Conf. on Innovative Data Systems Research (CIDR)*, January 2003.

[OJW03] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, June 2003. (To appear).

[SQR] SQR – A Stream Query Repository. http://www-db.stanford.edu/stream/sqr.

[Vit85] J.S. Vitter. Random sampling with a reservoir. *ACM Trans. on Mathematical Software*, 11(1):37–57, 1985.